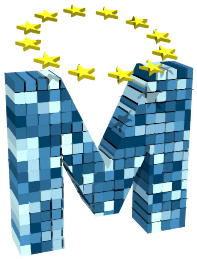

Project title: Multi-Owner data Sharing for Analytics and Integration respecting Confidentiality and OWNer control
Project acronym: MOSAICrOWN
Funding scheme: H2020-ICT-2018-2
Topic: ICT-13-2018-2019
Project duration: January 2019 – December 2021

D4.3
Final Encryption-based Techniques

Editors: Stefano Paraboschi (UNIBG)
Reviewers: Jonas Böhler (SAP SE)
Aidan O Mahony (EISI)

Abstract

This deliverable describes the tools that have been developed within Work Package 4 to support the use of encryption in the protection of data. The deliverable first illustrates the role that pseudonymization, encryption, hashing, and tokenization play in the protection of sensitive data. This analysis will be the basis for the realization of the tools supporting the policy-driven protection of data in digital data markets. The deliverable then describes advancements in the use of the Mix&Slice technique. First, the integration of the *aesmix* library with *File system in USEr space* (FUSE) is presented, which offers transparent file system access and service invocation. Then, the variant of the mixing technique based on the adaptation of *Optimal Asymmetric Encryption Padding* (OAEP) is described, with benefits in terms of flexibility, and also performance enhancements in some scenarios.

Type	Identifier	Dissemination	Date
Deliverable	D4.3	Public	2020.12.31

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825333.

MOSAICrOWN Consortium

1.	Università degli Studi di Milano	UNIMI	Italy
2.	EMC Information Systems International	EISI	Ireland
3.	Mastercard Europe	MC	Belgium
4.	SAP SE	SAP SE	Germany
5.	Università degli Studi di Bergamo	UNIBG	Italy
6.	GEIE ERCIM (Host of the W3C)	W3C	France

Versions

Version	Date	Description
0.1	2020.11.27	Initial Release
0.2	2020.12.15	Second Release
1.0	2020.12.31	Final Release

List of Contributors

This document contains contributions from different MOSAICrOWN partners. Contributors for the chapters of this deliverable are presented in the following table.

Chapter	Author(s)
Executive Summary	Stefano Paraboschi (UNIBG)
Chapter 1: Data wrapping for digital data markets	Michele Mazzola (MC), Stefano Paraboschi (UNIBG), Megan Wolf (MC)
Chapter 2: Integration of the Aesmix library with a file system	Michele Beretta (UNIBG), Sabrina De Capitani di Vimercati (UNIMI), Dario Facchinetti (UNIBG), Sara Foresti (UNIMI), Stefano Paraboschi (UNIBG), Matthew Rossi (UNIBG), Pierangela Samarati (UNIMI)
Chapter 3: AONT techniques for storage protection	Sabrina De Capitani di Vimercati (UNIMI), Dario Facchinetti (UNIBG), Sara Foresti (UNIMI), Stefano Paraboschi (UNIBG), Matthew Rossi (UNIBG), Pierangela Samarati (UNIMI)
Chapter 4: Conclusions	Stefano Paraboschi (UNIBG)

Contents

List of Figures

Executive Summary

The goal of this document is to describe the work done in MOSAICrOWN until M24 toward the development of tools for the use of encryption in digital data markets. Chapter 1, Chapter 2 and Chapter 3 each describe a separate aspect. Further work on additional components of the MOSAICrOWN architecture for the use of encryption is going to be reported in the next deliverable D4.4 due at M34, which will benefit from the work toward the preparation of Work documents W4.3 and W4.4, due respectively at M26 and M27.

Chapter 1 presents an evaluation of the use of protection techniques for digital data markets. It first presents pseudonymization, showing how data masking and data scrambling can be used to replace sensitive values with pseudonyms, without the use of cryptographic techniques. The chapter then describes the use of encryption, hashing and tokenization for the protection of sensitive data. In general, encryption, hashing and tokenization can be interpreted as techniques that permit a robust realization of pseudonymization. The chapter describes the main features of each of these techniques and illustrates the considerations that drive their use in concrete scenarios. Chapter 1 often references the financial domain, where one of the MOSAICrOWN partners is a central player. In general, security experts know that the construction of cryptographic algorithms is hard, both in their design and their implementation. The MOSAICrOWN project respects this common knowledge and assumes that the basic building blocks that will be used are cryptographic structures in common use, like *Advanced Encryption Standard* (AES) for symmetric encryption and the *Secure Hash Algorithm* (SHA) family of hash functions for hashing. The important innovation that MOSAICrOWN plans to work on is the construction of an architecture where the specific protection technique is applied based on requirements expressed in the MOSAICrOWN policy language. The analysis of Chapter 1 justifies the options that the tools will be able to support.

Chapter 2 describes the realization of a tool that facilitates the use of the Mix&Slice technique and of the associated *aesmix* library that was developed in Deliverable D4.1. The idea is to realize a virtualized file system that permits users and applications to benefit from the increased protection offered by the use of Mix&Slice in a transparent way. Each operation to a generic file is mapped to invokes the services of the *aesmix* library. The tool is built using the FUSE library. Support is offered for all the operations that characterize access to files. Experiments evaluate the performance impact, which proves to be acceptable for the expected profile of a digital data market. This effort shows the compatibility of the Mix&Slice protection technique with a new scenario, extending the support for Distributed Cloud Storage that was reported in Deliverable D4.2.

Chapter 3 presents a revision of the mixing structure that was used in the *aesmix* library presented in Deliverable D4.1. The work starts from the consideration of the use of Optimal Asymmetric Encryption Padding (OAEP) as an alternative realization of an All-Or-Nothing-Transform. The mixing structure based on AES that was originally used in the *aesmix* library offers good performance, but it has a limitation on the maximum size of the information unit that can be replaced (called a *miniblock* in Mix&Slice), which can be large at most half the size of the encryption block (i.e., 64 bits for AES). The use of OAEP removes this restriction. The investigation shows that a

revised structure of OAEP must be used in this domain. Chapter 3 describes alternatives for the realization and then reports on experimental results, which show, in addition to the greater flexibility, a performance advantage compared to the AES-based structure when a hardware-supported implementation of AES is not available.

1. Data wrapping for digital data markets

The design envisioned in MOSAICrOWN is that data can be protected in several ways, depending on the specific requirements of each data items. The project identifies two families of protection measures: those based on the use of data wrapping (typically relying on the application of encryption, usually reversible), and those based on data sanitization (typically relying on the application of transformations depending on the distribution of data, which reduce the information content in an irreversible way).

This chapter provides a compact description of the features of the main techniques that are envisioned for data wrapping. The idea is that these techniques will be applied on the data based on the specification that will be provided in the MOSAICrOWN Policy Language. The tools responsible for this policy-driven application are being developed and will be described in the deliverables produced in the next period by Work Package 4. The use of a policy-driven protection offers the flexibility to select various data wrapping techniques. The variety of techniques offered from the system will provide a level of security additional to the classical access control solutions implemented by the system managing the data in the market. The toolchain is expected to handle any combination of wrapping techniques.

The major goal of the application of these techniques is to protect sensitive data in such a way that data can no longer be attributed to a specific subject, anonymizing them. Additionally, these techniques help protect data in transit and at rest. All of these techniques, in use together, significantly reduce the risks associated with data processing, while maintaining data utility.

The techniques we are presenting are pseudonymization, encryption, hashing and tokenization.

1.1 Pseudonymization

Pseudonymization, at a high level, is the switching of information that can identify a given subject to non-identifiable data. The key to the application of pseudonymization is the determination of direct and indirect identifiers.

Direct identifiers are pieces of data that can be used to identify a person by cross-linking with publicly available data. The most common example of a direct identifier would be a national identification number. This data, when compared to indirect identifiers, are far more critical to ensure overall security, and are typically the first focus of any pseudonymization efforts.

Indirect identifiers, on the other hand, are pieces of data that do not identify an individual in isolation, but can be used to identify an individual when combined with other data. For example, if given a cardholder's nationality, it would be hard to determine the individual, but in conjunction with a card number, a name, or some other piece of information, it would be possible to identify the cardholder. Indirect identifiers are less critical to securitize.

The direct identifiers must be protected, and a common option is to use a pseudonymized value that is used in replacement of the secure data. The nature in which the replacement value

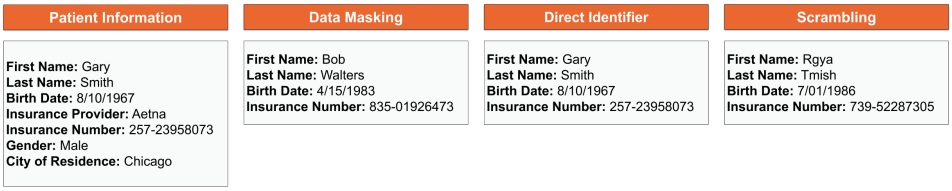

Figure 1.1: Example of pseudonymization techniques

is generated varies depending upon the pseudonymization technique used: data masking, scrambling, encryption and tokenization are all forms of pseudonymization. The latter two will be discussed in subsequent sections. The former two, data masking and scrambling, produce realistic outputs based on the input data. Data masking is the replacement of the real values with values extracted from the same domain. Data scrambling shuffles the elements of the real values to build a pseudonym with some similarity to the original value.

For example, consider a hospital wanting to pseudonymize patient information [NH11]. They would first have to determine what data is a direct identifier, and thus necessary to pseudonymize, then choose the corresponding technique. If the hospital wanted to use data masking, it would replace a name with a generic name with a 1-to-1 output to input relationship, as visualized below. However, if they instead decided to pseudonymize patient information using scrambling, they would simply jumble the existing letters to create a different output. Scrambling is obviously a weak form of anonymization. Figure 1.1 shows examples of pseudonymization.

1.1.1 Current use cases of pseudonymization

Many regulatory mandates have begun to recommend and require pseudonymization of data. This is mainly due to the fact pseudonymization reduces risk of data breach while still enabling meaningful data analysis. When direct identifiers are replaced with one or more pseudonyms, risk of personal identification, concerns regarding data processing and fears over data security are mitigated [HKN12].

Pseudonymization is currently used in many hospitals and organizations to introduce an additional layer of security to data without rendering it inaccessible for analysis or re-access [NH11]. For example, many Human Resource databases utilize data scrambling or data masking in annual employee career review processes. Additionally, when used in conjunction with other data wrapping techniques, as will be covered in the following sections, pseudonymization can further protect sensitive data from attack.

The main benefit of pseudonymization is the relative lack of impact on overall data analysis. Realistically, one would not need access to the pseudonymized data in order to make meaningful conclusions on data. For example, if a cardholder's name, card number and street address were pseudonymized, but other transaction level data was left untouched, analysts could still determine overall spending in a ZIP code or perform analysis on other transaction-level data without exposing the cardholder's personal data. It is for this reason pseudonymization is a central data wrapping technique, with interest for issuers, acquirers and merchants looking to securitize their data without impacting their overall ability to create meaningful data analysis.

However, despite the increased security pseudonymization introduces to data, there are some forms that can be reversed. Data masking and scrambling, when compared to data wrapping techniques based on encryption, hashing or tokenization, are relatively simple and often offer

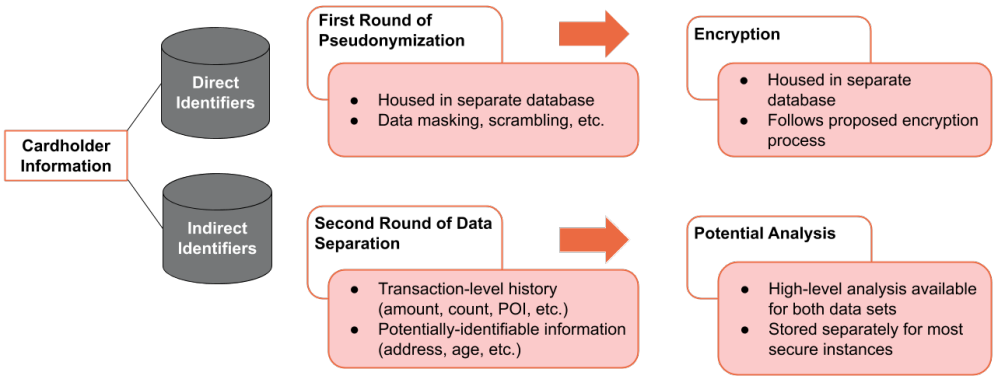

Figure 1.2: Pseudonymization plus encryption

limited protection.

1.1.2 Pseudonymization implementation recommendations

Given the ease of data breach for certain pseudonymization techniques, it is recommended that pseudonymization is applied using the robust approaches that will be discussed in the next sections. An option that can be sometimes considered is to couple simple pseudonymization techniques with either encryption or tokenization, to create double layered protection.

In situations that require high levels of security, the combination of multiple forms of data wrapping can be recommended. However, it must be noted this can be costly, both from a monetary and overall maintenance perspective. An increase in the use of securitization techniques leads to a growth in the amount of required storage space and in the complexity of maintenance and upkeep. Keeping that in mind, though, it is important to note that layering data wrapping techniques has proven to be highly efficient and effective [Tow10], and can greatly decrease the overall risk of data breach.

As an example, consider a high-profile, internationally recognized commercial goods store with a strong eCommerce presence. If cardholders decide to store their payment information to ease the payments process online, it is critical this information is stored securely. In this case, among the options for data wrapping, we can consider two options: pseudonymization plus encryption and pseudonymization plus tokenization, as described next.

Pseudonymization plus encryption

In this scenario, cardholder information would go through the pseudonymization process by specifying all the direct identifiers. From there, either data masking, scrambling, or any other pseudonymization technique could be implemented on direct identifiers, which then would undergo a second round of data wrapping through encryption (to be discussed further in Section 1.2). As an additional precaution, indirect identifiers could be further separated into transaction-level history and potentially identifiable information (i.e. ZIP code, city of residence, age, etc.). Figure 1.2 details this process further.

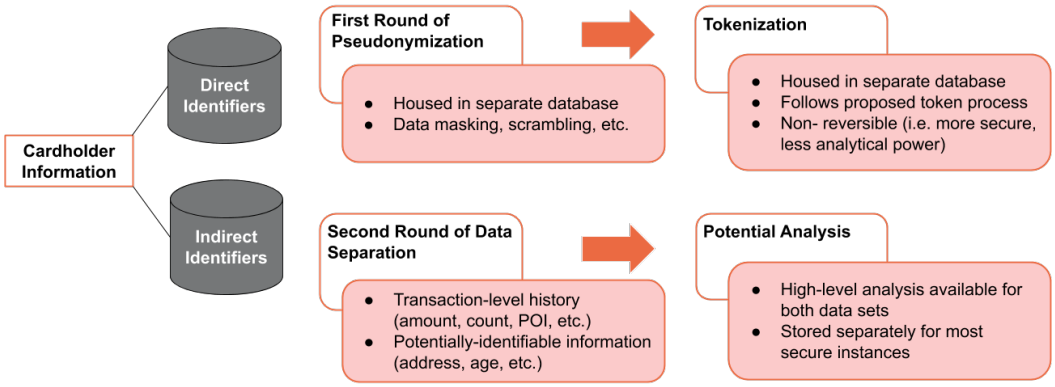

Figure 1.3: Pseudonymization plus tokenization

Pseudonymization plus tokenization

Similar to pseudonymization plus encryption, with pseudonymization plus tokenization direct identifiers would first be specified, then they would be pseudonymized followed by the application of tokenization. The key distinction to note is the fact that the correspondence between the pseudonym and the starting values is produced by a random process and is stored on a separate table. Since it is not protected by a cryptographic algorithm, the threat profile is different. For encryption, the major threats may be represented by adversaries with large computational capacities or knowledge on how to subvert a cryptographic algorithm or the ability to get access to the compact encryption key. For tokenization, the threat is only the access to the vault. Figure 1.3 gives an overview of potential implementation.

1.1.3 Analysis of pseudonymized data

While the main factor in implementing data wrapping techniques is data security and complying with regulatory requirements, it is important for issuers, acquirers and merchants to be able to derive information and key analytical findings from the stored data. It is critical in this process to ensure that, while keeping data secure, analysis remains feasible and accessible to issuers, acquirers and merchants, or else they will seek alternative resources that are not as thorough or focused on protection of data [NS19].

To reiterate, pseudonymization should rarely be used as a stand-alone data wrapping solution and should be used in conjunction with the techniques that are being described in the next sections.

1.2 Encryption for data protection

1.2.1 Encryption for data in transit and at rest

Encrypting data has long been the preferred form of protecting data, especially when applied prior to transmission or storage. Data encryption protects sensitive data as it is stored and as it is transmitted over a network. It can be used alone for a secure layer of data protection. However, used in conjunction with other data wrapping techniques, it creates an even stronger protection.

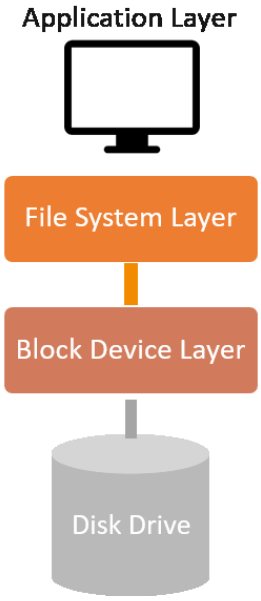

Figure 1.4: The four components of data storage

Encryption with data in transit

Data in motion is considered less secure and more vulnerable to attacks. Data must be encrypted when transmitted across networks to protect against eavesdropping of network traffic by unauthorized users. This application of encryption is invisible to the user and operates independently of other encryption processes, meaning data is encrypted only while in transit. An option is to use Internet Protocol Security (IPSec). A more flexible and commonly used solution is represented by SSL/TLS protocols. In some cases, isolation of the entire communication channel can be adopted, through the use of VPNs, which secure a communication channel between two networks with the use of encryption. While network to network data transfers are a common use case for encryption, encryption can also be applied to data sent between databases in the same segment, as well as data being sent over the internet. In all these cases, the data should be encrypted using a cryptographic algorithm such as AES. Data encryption in transit almost universally uses symmetric encryption, as it is faster and more efficient than asymmetric encryption. Asymmetric encryption has an important role in some scenarios, like in the use of *Public-Key Infrastructures* (PKIs), and this may lead to *hybrid encryption* architectures, where asymmetric encryption is used to establish a session key, then adopted as a secret by a symmetric encryption algorithm.

Encryption with data at rest

The role of data encryption when data is at rest is critical to protection of sensitive data. Data at rest generally refers to data that is stored in databases, the cloud, computer hard drives or even mobile devices. There are four components to a storage system as shown in Figure 1.4. The top most component is the application layer that accesses the file system through an interface. Then there is the file system, which is responsible for organizing files into directories on the disk followed by the access to the actual disk device. The access to the mass memory device is organized in blocks. Finally, there is the actual disk drive where the stored data resides.

Within MOSAICrOWN, three methods are considered to encrypt data at rest. They are full disk encryption, file system encryption and database encryption (i.e., the application of protection

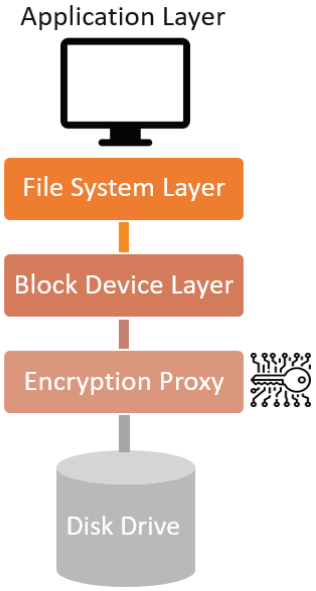

Figure 1.5: Full disk encryption

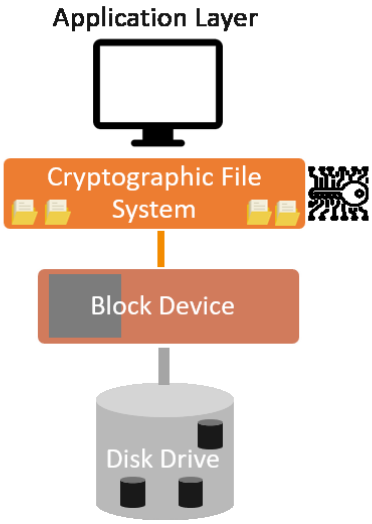

Figure 1.6: File system encryption

at the block layer supports the protection of the full disk content). Full disk encryption enables encryption of the entire disk including swap files, system files, and hibernation files. Disk encryption uses a symmetric encryption where a password is provided when the system boots to allow access to the data. With full disk encryption, the software installed on the server will not be impacted and can continue to be used as normal, since the OS provides transparent access to the encrypted data as necessary. Figure 1.5 shows where full disk encryption occurs.

System encryption, alternatively, allows the flexibility of encrypting only selected file systems or even just selected directories within file systems. This makes it possible to configure a server that requires access to a cryptographic key only when in need of access to the encrypted file systems. As with full disk encryption, file system encryption is transparently provided to applications by the OS. Figure 1.6 shows where file system encryption occurs. This approach is the one described in Chapter 2 of this deliverable.

The final method to encrypt data at rest is at the database level. This can be directly supported by the database server, or it may be managed by the application. The granularity in the application of the encryption can occur at several levels: the whole database, a single table, or even attributes or tuples within the database. Access to an encryption key is needed to access the protected data. The main focus of MOSAICrOWN is to offer tools that support the application of encryption at the granularity of distinct attributes, following the requirements expressed in a declarative policy.

1.2.2 Commonly used encryption algorithms

Two main types of encryption techniques are commonly considered, *symmetric* and *asymmetric*. Symmetric key ciphers use the same secret key for encrypting and decrypting a message or file. While symmetric key encryption is much faster than asymmetric encryption, the sender must share the encryption key with the recipient before it can be decrypted. Symmetric encryption algorithms require both parties to know the key or exchange the key via a secure channel. The most common symmetric algorithm is Advanced Encryption Standard (AES). AES offers 128, 192, and 256 bit encryption. AES is widely used because it is fast, safe, and often available in hardware implementation in modern CPUs.

The other type of encryption available, asymmetric cryptography, uses at least two different keys, often managed as one public and one private. Even though asymmetric encryption is slower compared to symmetric encryption, it offers many benefits. Organizations find themselves needing to securely distribute and manage massive quantities of keys, and asymmetric cryptography permits to distribute public keys without the need to protect them. Asymmetric cryptography allows for flexible message authentication, digital signatures and detection of tampering. A delicate issue is the management of the association between the identity of a user and its public key. Common asymmetric algorithms are RSA and ElGamal. ElGamal applied on elliptic curves is the foundation for *Elliptic Curve Digital Signature Algorithm* (ECDSA), which is today one of the most used solutions for digital signatures.

1.2.3 Key management

As the regulatory requirements for data privacy, confidentiality and protection have expanded and the need for protected data increases, the number of encryption keys that need to be maintained has greatly increased. The key management life cycle has evolved over time supported by innovations in hardware, technology and automation. For example, Hardware Security Modules (HSMs) enable the generation, safeguarding and management of truly random keys, which are needed for strong cryptography. They complement software solutions that facilitate the realization of adequate key management services. User and application access to these key management resources must be controlled and audited, so that data is never put at risk. These systems are particularly important when requirements impose to regularly change, cycle, and renew encryption keys. In this area, MOSAICrOWN aims at using best practices and tools that are already available for this goal. Several European research projects have developed solutions for these functions and these solutions can be candidates for a sophisticated realization of this important service.

1.2.4 Data protection and Homomorphic Encryption

Traditionally, a major feature of encryption is the emphasis on knowledge of the key: those who have access to it are able to decode and use the securitized data; those who do not have access

to the key cannot operate on the data. However, with the increased usage of cloud services and third-party service providers, the opportunity increases to design protection techniques that allow a user who has no access to the key to still operate on the encrypted data in a meaningful way. Homomorphic encryption is the major representative of these approaches, as it allows a third party to operate on the encrypted data without having access to or knowing the contents of the decrypted data.

Homomorphic encryption (HE) is a cryptosystem which allows arithmetic operations such as addition and multiplication over encrypted data without decryption. HE is a promising solution which prevents private information leakage during analysis on sensitive data.

Homomorphic encryption has a multitude of potential applications, from analyzing medical data to enabling private queries in search engines. There are three types of homomorphic encryption:

1. Partially Homomorphic Encryption (PHE): only allows selected mathematical functions (typically, only addition or multiplication) to be performed on encrypted values. One operation can be performed an unlimited number of times on the ciphertext.

2. Somewhat Homomorphic Encryption (SHE): supports limited operations up to a certain complexity, but these operations can only be performed a set number of times.

3. Fully Homomorphic Encryption (FHE): it is capable of using any efficiently computable function any number of times. It can also handle arbitrary computations on ciphertexts.

FHE is then more powerful, but current schemes are too expensive in terms of computational cost (the first implementations executed simple operations in several minutes; progress has been made, but operations are still requiring a cost that is several orders of magnitude greater than that associated with operations on plaintext data). Within MOSAICrOWN, the goal is not to develop new encryption algorithms, rather to configure their use in a way that is consistent with the protection requirements. Homomorphic encryption schemes are part of the solutions adopted by the systems.

1.3 Hashing

This section describes the use of hashing in data protection. Though there are many different techniques that can be used to protect sensitive data, there are scenarios where hashing proves to be the best fit. Section 1.3.1 will focus on the basics of hashing as they pertain to data wrapping with MOSAICrOWN. Section 1.3.2 will discuss when hashing and its one-way transformation would be useful and will comment on the concrete algorithms to use.

1.3.1 The basics of hashing

Hashing works as a function that produces a *digest*. The input can be of arbitrary length, whereas the output has a length that only depends on the algorithm used. In mathematical terms, it is very simple:

$$H : X \rightarrow Y$$

where H is the hashing function, X is the set of possible inputs (strings of bytes of arbitrary length) and Y is the set of possible outputs (strings of bits of fixed length).

The first property of a cryptographic hashing is to implement a non-invertible function, i.e., given the output of the function, it must be computationally infeasible to identify an input able to produce that output. Another related property is to be collision-resistant: it must be difficult to find two different inputs $a \neq b$ such that $H(a) = H(b)$. Another property of a cryptographic hashing algorithm is ensuring the *avalanche effect*. This property ensures non-linearity of outputs: regardless of the size of a proposed change in input, the output from the hashing algorithm should change significantly. A change in a single bit of the input must have the opportunity to change, with 50% probability, each of the bits of the output. This is needed to ensure a secure, difficult to trace output.

It is then possible to use hashing to protect data by replacing the values of any sensitive attribute with the result of the application of a hashing function to the values. This simple approach can be applied in many cases. There are cases where this approach may still leave the data exposed. E.g., if the same hashing function is used for all occurrences of the data, then it may be possible to recognize when the same values occur and use this to re-identify portions of the data. Due to this concern, and the increasing regulatory restrictions on data wrapping techniques, many hashing implementations have started to include a *salt* (or *nonce*). A salt is a randomly chosen value that is used as an additional input when computing the hashing function. Identical values with different salts are mapped to different outputs. With the use of a salt, protection is offered against the use of rainbow tables (structures that allow to determine an input for a given output of a hash function, built with large computational resources against a restricted input domain).

Also, if the hashing function is known and it is possible to make guesses on the protected values, access to the hashed values may let an adversary have access to the protected data. In this case, the use of salts that are not visible can provide protection against this attack.

A key component to a strong salt is ensuring it is dynamic and fresh for every use. If the salt was static, identical values would still map to the same hashed values. While using a static salt makes dictionary attacks more difficult, a dynamic salt is a more robust solution. A consequence of the use of dynamic salts is the reduced ability to execute analysis on the protected data. The selection between the use of direct hashing without salts, hashing with static salts, and hashing with dynamic salts must be done after a careful consideration of the risks and benefits of each solution.

1.3.2 Scenarios for hash implementation

Hashing for data protection in scenarios where sensitive data must be processed has gained rapid traction in the past two decades. In this section, we will consider the impact applying hashing will have on issuer, acquirer or merchant data analysis, as well as the most recent adaptation of this data wrapping method.

Hashing and analysis potential

Hashing is a one-way function, so it is often used when data does not need to be re-identified, thus it is most commonly used with sensitive data that is not required for specific system processing or analytics. Examples of these include password storage, card numbers that are not necessary for analysis on a large scale, and sensitive personal data that does not impact overall analysis, among other situations. It is important to note that, given increased privacy concerns regarding personal data in the public space, financial applications may find hashing beneficial for issuers, acquirers and merchants to appease the cardholder. Realistically, a full card number is not necessary for all

analysis. In situations like these, hashing is a cost-effective and intelligent solution to securitize data.

Further, in situations where the frequency of values or cardinalities need to be calculated without any additional information, hashing is an adequate data wrapping technique. Since hashing (without dynamic salts) is deterministic (i.e., it provides the same output for the same input), overall cardinality counts will remain unaffected.

Hashing should be considered as a data wrapping technique for those data items that may not require enhanced analysis methods. In the portfolio of securitization methods, hashing is a secure and efficient way of gaining high-level data views without exposing sensitive values. This one-way protection method ensures that not even those storing the data will be able to see the protected data. While those who store the data may be able to see other elements of a transaction that can be used to further analyze and draw meaningful conclusions, critical elements of personal identification, like card number, name, or zip code, can be hashed to ensure privacy and compliance with regulations. For merchants, especially, hashing can be useful on a multitude of levels, such as, providing high level findings while protecting names, or stored card numbers. In general, the use of hashing can reduce risk of breach in conjunction with providing possibility for meaningful data analysis.

In terms of concrete algorithms, the family of hashing functions proposed by the National Institute of Standards and Technology (currently SHA-2 and SHA-3) are adequate in terms of performance and security for all the scenarios where hashing is going to be used.

1.4 Tokenization

This section discusses tokenization, which has been widely implemented in the payment card industry due to regulations in this industry, but started to have use cases in other industries as well.

1.4.1 Tokenization for data protection

Tokenization, or early forms of it, have been in practice since a long time to protect sensitive data. In the digital age, it has become standard practice to use tokenization to securitize the storage of credit card numbers and hence has gained traction in protecting sensitive personal data.

The tokenization process is simple in its conception: the data is split into various input sections, which are eventually replaced by dummy variables, or tokens, to securitize the data. These tokens are produced in a randomized, unpredictable way, making the secure data hard to re-identify without having access to the tables where the mappings between tokens and original values are stored. In the end, the tokens replace the input data with a 1:1 mapping, to create a new output composed of randomized digits, independent of the secure data. These 1:1 mappings are stored in the *token vault*.

Two components of this process are worthy of further consideration. The first is the initial creation and storage of the tokens. To illustrate, consider a card on a mobile wallet. Figure 1.7 shows the architecture overview of taking a non-tokenized card number, storing it on a mobile phone, and subsequent retrieval of the card information in payment flows [MR19].

The second component is the use of tokenized data to access secure data. Consider a mobile card payment. The card holder presents their mobile card at a Point of Sale (POS). At that point, the system within the mobile phone sends the tokenized card number through the centralized pay system. The tokenized card number travels through the acquirer to the token vault, where an

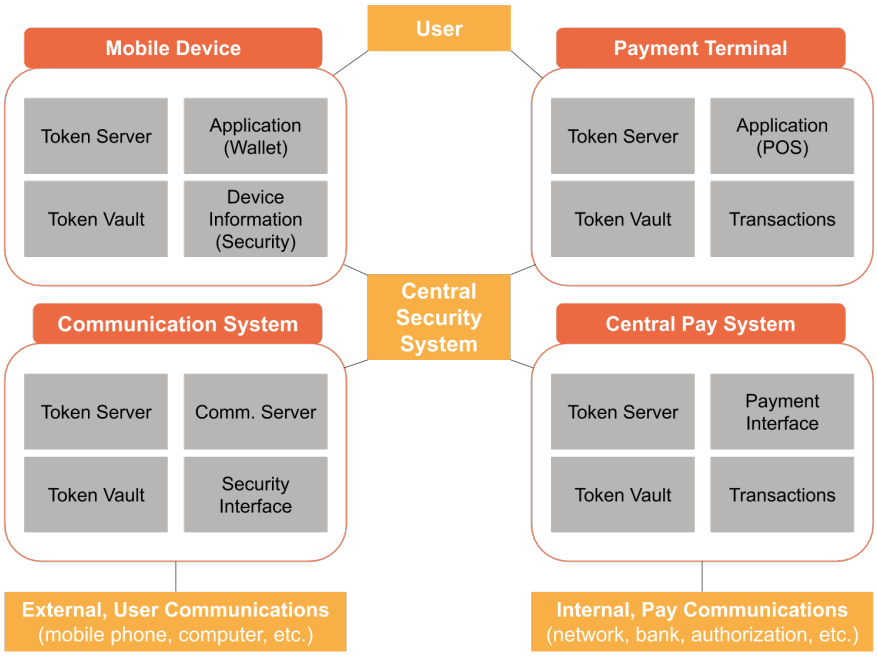

Figure 1.7: Example of tokenization process in payment industry

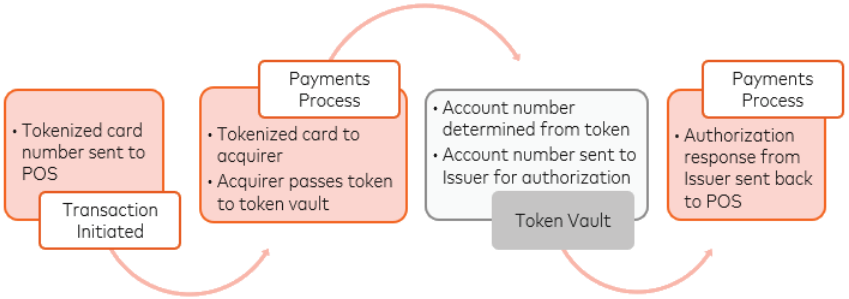

Figure 1.8: Tokenization steps

account number is determined by mapping the tokenized number to an account. From there, the payment process proceeds as normal, and the issuing bank is alerted and determines if the payment should be processed [MR19]. The key of this illustration (Figure 1.8) is as follows: every step that has potential for fraud is bypassed before the token is re-mapped to key data.

It is clear the main benefit of implementing tokenization is the security it provides to data. However, in order to reap the benefits of this security, it is critical to use strong randomization practices when using and implementing tokenization. Simplified tactics, like non-salted hashing, are inadequate, as non-salted hashing is deterministic and exposed to dictionary attacks, as discussed in Section 1.3. Ideally, the randomization should be impossible to reverse engineer. A robust random number generation is feasible and represents the most reliable form of tokenization.

On the other hand, one of the difficulties in using tokenization is accessing secure data for either analysis or reporting. In cases where data contributors need reference data, a form of tokenization can be used where tokens are created as a combination of random values along with a portion of the original data (e.g., last 4 digits of a credit card number). While not as secure as a completely randomized token, these portioned tokens are effective for referencing purposes.

Above all, the randomization and lack of inherent value the token holds is what makes using tokenization as a wrapping technique secure. Furthermore, sensitive data will never enter or reside within the shared data platform: regardless of the secondary data source and its security, the true data can never be obtained. Not only is the secure data isolated from any secondary data source, but the lone identifier that can be used to reverse the tokenization is also independently and exclusively stored on the secure platform.

Before proceeding, it is important to note there are scenarios where tokenization is not the ideal wrapping technique to utilize. Tokenization can be prohibitive to implement, as maintenance costs and scalability of data storage can hinder initial instance set-up and long-term practice. Further, third-party sharing of tokenized data can be cumbersome as direct access to token mapping would be required.

1.4.2 Token vault management

When considering tokenization, one must determine specifications of the token vault. There are three specifications to consider:

1. Location
2. Fluidity
3. Reversibility

Location of the token vault

There are two options for location of token vault, either engaging with an external token service provider or storing the tokenization data internally. While it may seem counter-intuitive to engage a third-party system to increase security in the management of sensitive data, the former option allows for a hands-off, potentially more cost-effective alternative to implementing protection in-house. The latter allows for complete control over the tokenization process, and ensures all aspects are internally managed and maintained, resulting often in higher costs for both implementation and maintenance.

However, choosing location can impact the other two specifications: if tokenization is outsourced, it must be reversible for the initiator to continue to receive critical data. For example, if a bank outsources its tokenization efforts, in order to see any payments related data that would be of use, the tokenization efforts would have to be reversible, since the bank does not host the token vault.

Fluidity of the token vault

Fluidity refers to the nature of the token itself: is the table static or dynamic in generating tokens? If it is a static lookup table (SLT), it generates a hard-coded token that is used every time for the secure input. The SLT maps each component of the secure input to a token, and that mapping is stored in a table which is then used indefinitely for the lifetime of the token.

Alternatively, and increasingly utilized, the dynamic lookup tables (DLT) approach generates a new token every time the secure input is called for. Rather than storing the tokenized value, the DLT holds seed values, which then generate perfectly randomized outputs when activated. Albeit more secure, DLT's require far greater storage and larger data transmission per use.

Reversibility of the token vault

Token vault reversibility is simply the ability or inability to reverse the tokenization process. As previously stated, if tokenization is outsourced, the token vault must be reversible. However, there are other instances in which reversibility could be beneficial. When data has the potential to be shared with a third-party, or if it will need to be unwrapped later, reversibility is crucial. Clearly, non-reversible token vaults are more secure, and less likely to have breaches, but are also rarely used due to the limitations it imposes on data analysis.

Solidifying the characteristics of a token vault is the first step of implementing a strong tokenization practice. This vault will manage all aspects of the token process, from generating the token requests, to authentication, token storage, and detokenization. It is important to note that, as third-party contributors send data to the token vault and the vault begins to interact with the third-party data systems, it will need to be able to integrate using identity, access management, etc. However, the token vault is the most crucial step to securitizing data through tokenization, thus it should be fully encrypted and protected to ensure successful data protection.

1.4.3 Tokenization and encryption

Encryption and tokenization are often used synonymously; however, there are a few key differentiators between the two. Encryption transforms data to be unreadable by those who do not have access to the encryption key, while tokenization inherently changes the value of the data to be otherwise unrecognizable. While tokenization is recognized as the more secure way to wrap data, the two methods can be combined. This is most common with reversible tokens, as it increases the security and complexity of the storage. Encryption can be inserted into any point of the tokenization process, meaning it can occur before, after, or potentially even during token generation. Specific implementations vary depending on use cases, however, the combination has been successfully implemented in various instances. The tools for the policy-driven application of protection to sensitive data developed in MOSAICrOWN will consider the specification of this combination.

It must be noted that, if the combination is used, encrypted data, specifically the encryption key, must be stored separately from the tokenization data. Typically, these data sets reside in separate servers to avoid having a single point of failure.

1.4.4 Recommendations on implementation of tokenization

While the overall technology in implementing tokenization is simple (a random generator and the management of the token table), the level of complexity within each step is important in the overall implementation process. Given the various options in creating a tokenization process, it may be clear that there are differing levels of security and situations in which tokenization might be implemented. The most secure and widely used are token vaults that are internally housed, with a dynamic and irreversible token. However, there are various impediments to enforcing these recommendations on a large-scale, it is not realistic to expect all types of companies, large and small, to have the same ability to successfully create and maintain storage for this large amount of data [Jan13]. If it is not critical to tokenize data for security purposes, it may be a smarter, more cost- and space-effective decision to choose a different data wrapping technique.

With that being said, for companies that either have the data storage and resources to internally house their tokenization space, or for those who do not need access to the tokenized data and are willing to allow a cloud provider to house the tokenized data, tokenization is an important option.

As outlined throughout this section, the main consideration that motivates the use of tokenization is a limited need to access the protected data. With options like keeping the final four numbers of a credit card number de-tokenized, tokenization can securitize cardholder personal information without completely limiting the ability of an issuer or acquirer from deeply analyzing the data. Further, large payment providers, like Mastercard, continually work towards improving their tokenization process.

As the world continues to see an increased adoption of data protection techniques, it is clear that the accessibility and security of tokenization is a large benefit. However, in order to truly reap these benefits, tokenization must be carefully and methodically implemented and maintained. MOSAICrOWN has the ability to massively scale tokenization in partnership with a company that values continually improving the technology to be as secure, efficient and effective as possible.

2. Integration of the Aesmix library with a file system

The Aesmix library implements the Mix&Slice technology and makes it available for a variety of scenarios. The structure and main features of the library have been described in Deliverable D4.1. The theoretical foundation of the proposal and its application to a Distributed Cloud Storage scenario has been reported in Deliverable D4.2. The goal of the tool described in this chapter is to demonstrate the use of the Aesmix library in a generic file system. The file system interface is the one most commonly used for the access to the storage service. Offering a way to transparently use the Mix&Slice approach when a user or application directly accesses a file, greatly facilitates the use of these functions. The advantages that are provided in terms of better support for access revocation and secure deletion are then available in a more direct way.

The implementation and then the experiments on the use of the tool, reported in Section 2.4, demonstrate the limited performance cost associated with the use of the library when reading or creating files. The experiments show that the execution of write operations that require to modify only a portion of the file are delayed, but this is consistent with the profile of the technology, which is based on the use of an All-Or-Nothing-Transform. This realization then makes clear the scenarios where the approach is particularly suited, i.e., domains where the data are subject to read and create operations and the frequency of update operations is low. The digital data market domain that is at the center of the research in MOSAICrOWN is such a domain, as the data collections that are contributed to the market are in many cases stable and not expected to be subject to continuous updates.

2.1 Scenario

The file system is the operating system module that takes care of information management on mass memory, which is any peripheral device other than central memory. This is one of the most important functions of an operating system. In particular, the file system must:

- provide a unique and permanent identification mechanism for files, so that they can exist and be accessible for a long time;
- provide access methods that allow addressing, reading and writing elementary blocks of information contained in the files;
- mask the physical characteristics of the storage units;
- implement access control mechanisms on files, both to ensure confidentiality of information and to avoid concurrency issues;

- guarantee the permanence and consistency of information even in the case of hardware or software malfunctions.

The main data structures a file system works with are *file descriptors*. Descriptors contain various properties of the file, the main ones being the name of the file and its address, i.e., its position in mass memory. A set of descriptors is called a directory. Directories also have to be stored on disk permanently and they are handled in Unix-like systems like files: more precisely, in the Unix system a directory is a file that contains the list of names and addresses of the files contained within it.

In order to access a file, its descriptor must be obtained from mass memory. However, this operation is relatively heavy in terms of the number of disk accesses, particularly if it has to be repeated for every operation on the file. Consequently, the *open* primitive is introduced: before any operation on the file, its descriptor is retrieved and its content transferred to main memory. This descriptor will then be used to efficiently read and write to the file. When no further operation needs to be performed on a file, the inverse *close* operation is executed: the image of the descriptor contained in main memory is copied to disk, so to update the information contained in the file. We describe the opening of a file as the action of searching and copying in memory the descriptor of a file (and possibly its content), and the closing of a file as the release of the memory dedicated to the descriptor, resulting in the transfer of its content to disk.

2.1.1 Virtual file systems

A virtual file system is a software component that allows the operating system kernel to access the file system through the use of functions independent from the actual file system (or the device used for storage). Access to files stored on other computers via the network, or access to files encrypted without the user noticing the encryption, are examples where the virtual file system comes into play and simplifies the interaction between the user and the computer.

FUSE (*File system in USErspace*) is a software system that allows the creation of virtual file systems on the Linux kernel [FUSa]. The FUSE system consists of two parts: (1) the kernel module for communicating with the Linux kernel; (2) the *libfuse* user library [FUSc], which implements communication between the virtual file system and the kernel module. FUSE therefore allows the user to directly adopt the virtual file system on Linux and, in general, on Unix and Unix-like systems (including FreeBSD and macOS) thanks to a porting and rewriting process. Figure 2.1 shows how FUSE interacts with the Linux kernel. Every request made by user space (e.g., `ls -l /tmp/fuse`) is intercepted from the VFS kernel module (*Virtual Filesystem Switch*, a software layer of the Linux kernel that manages the interaction between userspace and the file system) and is redirected to the FUSE kernel module. The request is then passed to the *libfuse* userspace library, which runs a specific program configured to handle a request (`./hello` in the figure). The program response is then sent back to FUSE, and forwarded to the initial program in userspace.

2.1.2 Encrypted file system

An encrypted file system is a special type of file system capable of handling encrypted files. In order for a user to access files, it is necessary to provide to the file system the key used for encryption. The benefits of an encrypted file system are essentially the following:

- *confidentiality*, as the files are encrypted and are not accessible to users who do not have access to the encryption key;

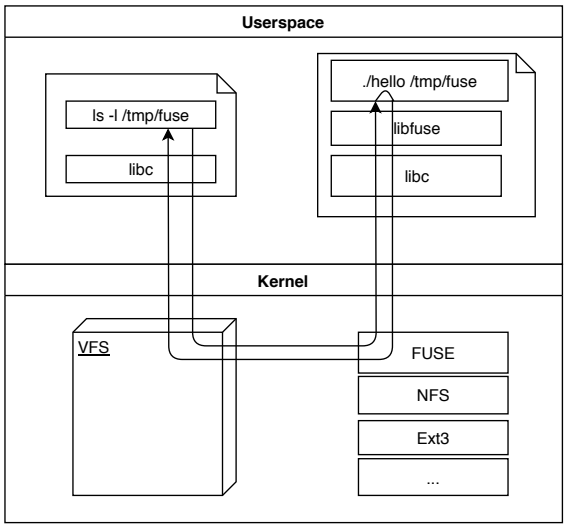

Figure 2.1: FUSE architecture

- *transparency*, as the encryption and decryption operations are not visible to the authorized user, who browses through files as the user would in a non-encrypted file system.

While accessing the stored data, the file system encrypts and decrypts files if necessary, keeping only the encrypted data on mass memory. In this way, an adversary who gets access to the physical storage medium would not be able to read the content of the files.

The most well-known example of the realization of an encrypted file system is *EncFS* (*Encrypted File System*), an open-source library released under the LGPL license, which implements a virtual file system with FUSE. EncFS encrypts the files individually and translates all requests for the virtual file system to equivalent calls to the operating system. In particular, there are some features that distinguish it from other virtual file systems:

- it has a "reverse mode", i.e., it provides an encrypted view of directories that are not natively encrypted;
- it is relatively fast on traditional hard disks;
- it also works with network file systems.

The goal of the tool described in this chapter is to take inspiration from the EncFS project and to implement a virtual encrypted file system using as encryption functions the whitebox-AONT functions offered by the Aesmix library, which was described in Deliverable D4.1.

2.2 Design

The virtual file system was implemented with the Python language. The platform used for the development and testing was the Ubuntu operating system, but the solution should work on all Linux distributions. Among the libraries used in the project, the two more important are *fusepy* [FUSb] and *aesmix* [MOS20].

The *fusepy* library [FUSb] is a Python module that offers bindings to the C implementation of FUSE for Linux and macOS systems. The *aesmix* library provides the implementation of the

Mix&Slice approach to encrypt and decrypt files. The *aesmix* library offers the Python *mixslice* wrapper, which permits to invoke from Python the services to encrypt and decrypt files using Mix&Slice. In addition to the encryption layer based on the use of symmetric cryptography, the *aesmix* library uses asymmetric cryptography to manage the access to the encryption keys. The *mixslice* wrapper accepts a file as input and outputs three elements:

- a directory containing the encrypted fragments produced by the invocation of Mix&Slice;
- a `.public` metadata file, which contains the public key;
- a `.private` metadata file, which contains the private key.

The main methods of the *aesmix* library that were used for this work belong to the `MixSlice` class and are the following:

- `load_from_file`, which loads the files needed for decryption into memory;
- `save_to_files`, which saves the many fragments and metadata files to disk;
- `encrypt`, to encrypt a file;
- `decrypt`, to decrypt a file.

The virtual file system needs three directories to be mounted and used:

1. a directory indicating the *mountpoint*, that is, the point where the virtual file system will be mounted virtual;
2. a directory containing the various encrypted data, possibly organized in subdirectories;
3. a directory that contains all the metadata files (`.public`, `.private`, and `.finfo`).

The directories for encrypted files and metadata files are separate, in order to allow generality in the use of the file system. The structure of both directories is the same to ensure uniqueness in the association between encrypted files and the corresponding metadata, as files with the same names can exist in several points of the file system. For example, if an encrypted file has the path `dir/image.png`, then its metadata must be `dir/image.png.public`, `dir/image.png.private` and `dir/image.png.finfo`.

Since the *aesmix* utility produces a directory containing the fragments, to distinguish between directories and encrypted files, the file system refers to the existence of metadata files or not: if the corresponding metadata exist, then the directory is treated as if it were an encrypted file, otherwise its management is left to the operating system.

2.2.1 Management of encrypted files

The file system must show the directories containing the encrypted fragments as files. To make this happen, it is necessary to change the information found in the file descriptor seen by the system. Since directories are files, their descriptor is retrieved in an analogous way as it is done for files. At the request of the descriptor by the system, the following properties are modified:

- `st_mode`: contains permissions and file type. To let encrypted directories appear like files, the `S_IFREG` flag (which identifies a file) is set, and the `S_IFDIR` flag (which identifies a directory) is removed. The other permissions remain unchanged.

- `st_nlink`: represents the number of hard links of the file. This is equal to 2 for a directory, then it is made equal to 1 to make encrypted directories appear as files.
- `st_size`: the size in bytes of the file.

The operating system uses the `st_size` field to know how many bytes to read from disk. Offering as file size the space occupied globally by the fragments in the directory could cause problems when reading the file, e.g., when using a GUI to access the file content. When the file is opened, if the size is not correct we may have the system reading fewer bytes than are contained in the file, therefore considering the file as corrupt, or reading more bytes than necessary, leading to crashes and forced closures of applications. Nonetheless, decrypting a file every time a descriptor request is made to determine its actual size can have a significant performance impact. For this reason, we introduced a new metadata file type, with the extension `.finfo`, which contains information about the size of the file in the virtual file system. This size is updated at each write operation, only if the size has actually changed. In this way, although with an initial overhead when writing to files, the file system is significantly more responsive in its normal use. What is described here can only be done if it is possible to open and decrypt the file and if there are any metadata files associated with it. The directories that do not contain the encrypted fragments are therefore processed directly by the system, which manages them without any intervention by the virtual file system.

Support for reading, writing, and creating files

In order to read and write a file, the file must be first opened, that is, decrypted and its content transferred in memory.

The reading takes place in blocks on the content in memory, through an offset and a total number of bytes to read. Writing, on the other hand, takes place via a buffer containing the data to be saved and an offset indicating the position of the byte to start writing from. It is also possible to truncate a file, or to limit its length to a certain value. On opening a file, a new value must be assigned to `st_atime`, which indicates the time instant of the last access to the file.

Writing has the particularity of not being carried out directly on mass memory, but it is done on the representation of the file in main memory. The file is encrypted on disk at a later time, when the blocks in main memory are flushed. For the sake of efficiency, the flush is done only if changes have been made to the file content, that is, if at least one write operation has been executed. In this way, if a file has not been changed since it was opened, no unnecessary encryptions are made, improving the efficiency of read-only operations. Writing also requires to change the `st_mtime` value, which indicates the instant (Unix timestamp) of the last modification of the file content.

Creating a file on disk corresponds to creating an area in memory with the necessary content, even empty. Furthermore, the support for file creation enables the ability to copy and duplicate files, as copying is essentially a creation followed by write operations.

2.2.2 Renaming and deleting files

Renaming an encrypted file also renames the associated metadata files. If done from a graphical interface, the operation does not allow a user to rename to an already existing file, while from the command line the renaming is indistinguishable from moving a file. The implementation of a correct renaming therefore enables both the ability to move files and the ability to overwrite files in the virtual file system.

Permanent deletion of a file also involves deleting the metadata files. Moving a file to the trash bin, on the other hand, is actually a move operation, therefore supported with renaming. If on the virtual volume created by FUSE there does not exist a directory used as a trash bin, this will be created. Even the files in the trash bin will be encrypted.

Neither the deletion nor the renaming need to open and close the files, therefore they are particularly efficient due to the absence of encryption and decryption steps. Both renaming and deletion, however, need to change the information kept in memory regarding various file characteristics, to avoid referring to data no longer valid (see the management of `st_size` and the implementation of *EncFilesManager*). It can also happen that a file is renamed while it is open.

2.2.3 Concurrency in opening a file

A file can be opened at the same time by multiple applications. At closing of any of these, the file system prevents a file from being closed if it is still in use by other programs, to avoid anomalies or data corruption. This translates into the need to maintain a counter for each open file in memory, which is increased at each opening and decreased at each closure of the file.

The file is finally closed, releasing the dedicated memory areas, when this counter reaches the value 0. It remains possible to perform the flush of a file to update its content on disk. The applications take care of updating the content of the file if this has changed on disk, or they offer the user the option to reload it. This happens thanks to the update of `st_mtime` during write operations.

2.2.4 Management of directories

To keep the exact same structure between the directory with the encrypted files and the directory with metadata files, the virtual file system must take care of replicating all directory operations (create, modify and delete) between the two locations on disk. This management does not present particular difficulties, except for the rename operation, as each operation is carried out on directories by the operating system through the same primitive used for files.

The rename operation is managed in a way different from the others: for example, to remove a file the *unlink* operation is used, whereas to remove a directory the *rmdir* operation is used. In the case of renaming, it is therefore necessary to differentiate between an encrypted file and directory through the `st_mode` property of the appropriately modified descriptor.

2.2.5 Multithreading

The Linux *libfuse* library permits to mount virtual file systems in two ways: (1) single-thread, where the entire file system runs on only one thread inside a process; (2) multi-thread, in which the various operations (writing, reading, etc.) are assigned each time to different threads.

The single-thread mode is less powerful from a performance point of view, but it is simpler and more suitable for development, since it requires no checking of conflicts among distinct concurrent threads. The multi-thread mode instead offers better performance, as it permits to parallelize file operations whenever possible, such as in the case of reading a large amount of data. Specifically, the *libfuse* library protects the critical sections of the code from concurrent accesses through semaphores, according to the POSIX standard, via the `pthread.h` library.

FreyaFS maintains these two paradigms, operating by default on a single thread. Although *libfuse* already takes care of the protection of critical sections at a low level, it is still necessary

to protect access to shared memory areas, mainly represented by the content of open files and counters of applications that have opened a given file.

The file system handles the actions of writing and reading of open files content differently. More readings can take place in parallel, in order to increase performance, while writes must not interfere with each other and must have exclusive access to the resource. This behavior is accomplished through the use of read locks and write locks. The virtual file system keeps for each open file a counter of how many threads are reading the resource: this is incremented every time a thread starts the read operation and decremented when it ends. When a thread requests write access to the file, it is put on hold if there are any readers present and is notified when there are no more. If instead the resource is free, access is guaranteed exclusively through a semaphore.

2.3 Implementation

Five files constitute the core of the project:

- `main.py`, which handles command line arguments and mounts the file system;
- `freyafs.py`, which manages the operations of the file system in general and communicates with the operating system;
- `encfilesmanager.py`, which implements the various input/output operations on files;
- `encfilesinfo.py`, which manages the `.finfo` metadata files;
- `filebytecontent.py`, which manages concurrent accesses to files.

Figure 2.2 shows the static structure of the code and how modules are related to each other. An arrow from module *A* to module *B* indicates that *B* uses module *A*. In the external modules we have the *aesmix* library, which implements Mix&Slice, and the *fusepy* library, which provides the implementation of FUSE for the Python language. The internal modules are those managing the encrypted files and realize the logic of the virtual file system. Finally, the *main* only manages the interface with the user from the command line and invokes *fusepy* to mount the file system.

2.3.1 Interaction with the user

The interaction with the user is managed by the `main.py` module, which is responsible to process the parameters passed on the command line and to mount the file system. The parameters that can be set are shown in Table 2.1.

Parameter	**Use**
`MOUNT`	The mountpoint, which is the path where the file system will be mounted
`--data DATA`	The path to the directory containing the encrypted files
`--metadata METADATA`	The path to the directory containing the metadata files
`--multithread`	Flag indicating whether to mount the file system in multi-thread mode (default to `False`)

Table 2.1: `main.py` parameters

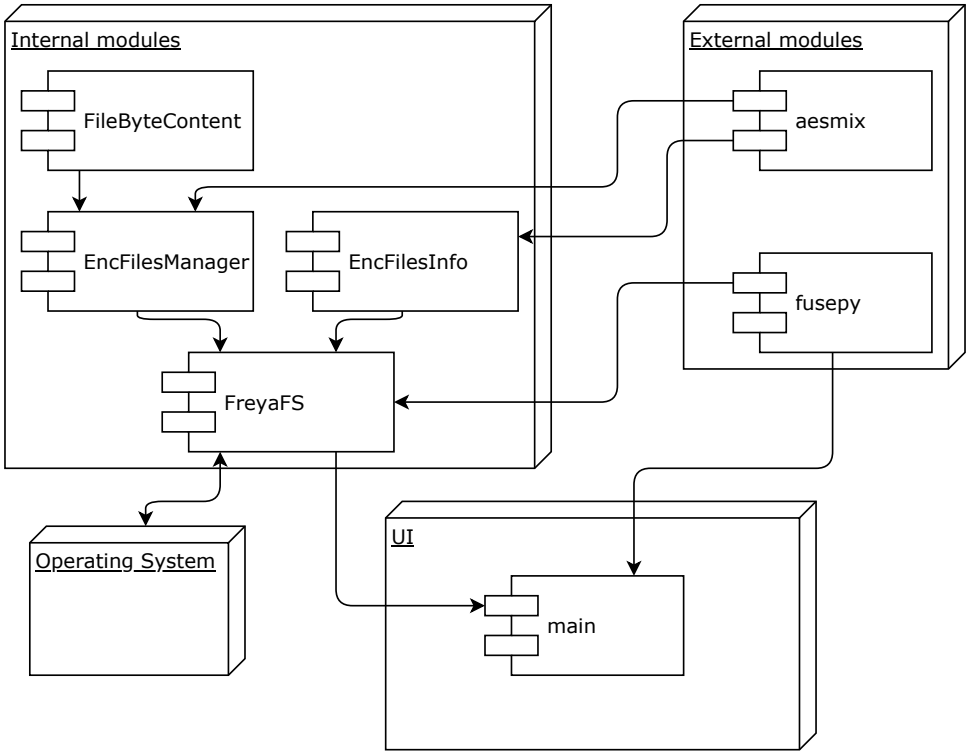

Figure 2.2: Structure of the code

2.3.2 FreyaFS

The *FreyaFS* module implements FUSE's methods for managing the virtual file system. In particular, the most important part of the module is the `FreyaFS` class, which extends the `Operations` class of FUSE and overrides some methods. Many of the methods of the class simply call the operating system, as they do not require encryption support (such as the test to access a file). Other methods are redefined to support the file encryption and decryption, performed by the *encfiles-manager* module.

The `getattr` method is used to get information on the file and directory descriptors. The method computes the size of the encrypted file and changes the flags `S_IFREG` and `S_IFDIR` to make the directory with the fragments appear as a single file. The computation of the size is done by instantiating the `EncFilesInfo` class, only if it was not already done previously (in order to improve performance by avoiding redundant decryption). The `self.enc_info` property is a dictionary that contains the various instances of `EncFilesInfo` for each encrypted file found in the system. Flags are changed to keep the same access permissions.

The `unlink` method supports the removal of files. In its code, only the primitives of the operating system are called, without opening or closing any files. Metadata files and memory areas dedicated to store file information are also deleted. The method needs to check the existence of the `.finfo` file, because, unlike the `.public` and `.private` files, it cannot exist as it is

Method	Goal
`getattr`	Retrieve the properties of an encrypted file or directory
`readdir`	Retrieve the content of a given directory, including the `.` and `..` entries, and ignoring metadata files
`mkdir`	Create a directory, both in the encrypted data path and in the metadata path
`rmdir`	Remove a directory, both in the encrypted data path and in the metadata path
`unlink`	Remove an encrypted file, also deleting the corresponding metadata files
`rename`	Rename an encrypted file (and corresponding metadata files) or a directory
`utimens`	Update the instants of the last update and the last editing for encrypted files and directories
`open`	Open an encrypted file
`create`	Create an encrypted file
`read`	Read a certain number of bytes from an open encrypted file
`write`	Write a certain number of bytes to an open encrypted file
`truncate`	Truncate the content of an open encrypted file at a given length
`flush`	Force writes made to an encrypted file from memory to disk
`release`	Release an open encrypted file

Table 2.2: Methods in the *FreyaFS* module

created on the first `getattr` call.

The renaming of a file, managed by the `rename` method, is more complex, as it is the same operation used for moving both files and directories. Furthermore, since encrypted files are mapped to directories, if the user wants to rename a file with the name of another file that already exists, the latter must first be deleted, to prevent the system from recognizing it as a directory and trying to move the first file inside the second one. Finally, the paths associated with the file and its metadata are also updated. For regular directories, i.e., those not containing encrypted fragments, the `rename` method directly invokes the operating system and replicates the update both in the encrypted data path and in the metadata path.

The `open`, `create`, `truncate`, `read`, `write`, `flush`, and `release` methods all have a similar structure. If an open file is not among those encrypted, then the file is managed by the operating system as it normally would. Table 2.2 summarizes the methods used by *FreyaFS* that were changed with respect to the basic case, where each method simply invokes the corresponding primitives of the operating system.

2.3.3 EncFilesManager

This module defines the class with the same name managing the input and output on encrypted files. Specifically, the class is instantiated only once and contains information on all files (encrypted with Mix&Slice) currently open, identifiable by their path in the real file system. It uses a counter for each open file that indicates how many applications are currently using it: this way, the class avoids to accidentally close a file if only some of these applications have released it while

others have it still open.

The methods offered by the `EncFilesManager` class are the following:

- `open`: opens a file, decrypting its content and keeping it in memory;
- `create`: creates an empty file, i.e., it reserves empty space in memory for the content of the file (only if not already open) and forces its flush;
- `read_bytes`: reads a certain number of bytes from an open file;
- `write_bytes`: overwrites part of the content of an open file with data contained in a buffer;
- `truncate_bytes`: truncates the content of an open file to a certain length;
- `flush`: writes the content of an open file to disk, encrypting it;
- `release`: releases the memory areas dedicated to a file, closing it;
- `cur_size`: gets the current length of the content of an open file;
- `rename`: renames the various paths referring to an open file.

It is essential to have a `rename` method, because a file can be renamed even if open: in this case, to avoid referencing files that no longer exist and cause the file system to behave incorrectly, it is necessary to update all the paths related to the file in question. The `EncFilesManager` class also tracks the change status of a file using Boolean flags. That way, if a flush is requested but there are no changes to the file, the encryption operation can be omitted, improving performance.

Finally, the `EncFilesManager` class also handles mutual exclusion for the multi-thread mode of the virtual file system, via a semaphore of the Python threading module that implements a lock. This semaphore guarantees exclusive access to structures containing the content of open files and open file counters.

2.3.4 EncFilesInfo

The *encfilesinfo* module defines the `EncFilesInfo` class, which is responsible of the management of attributes of the encrypted files that are not contained in the `.private` and `.public` metadata files. Its main job is to interface *FreyaFS* with `.finfo` files, which contain information serialized in the JSON format. Unlike the `EncFilesManager` class, the `EncFilesInfo` class is not aware of all the files managed by the virtual file system, but it only manages the information of a single file specified at initialization.

Specifically, the module supports the management of the length in bytes of the content of the encrypted file, which can be obtained and set using the property `size`. When the class is instantiated, the size of the desired encrypted file is calculated and is stored in the `size` property and also in the `.finfo` file (which, if not present, is created). When this property is updated, the value contained in the metadata file is also updated, only if the value is different from the previous one. This avoids opening a file, parsing the JSON contained in it, updating the property and then serializing the JSON produced, in case these operations are not strictly necessary. The class also provides a `rename` method, used to rename the path to the file containing this information.

2.3.5 FileByteContent

This module deals with reading and writing the content of a file opened in a mutually exclusive way. To ensure this, the `FileByteContent` class has the following properties:

- `_text`, which represents the content of the file as a Byte string;
- `_readers`, which indicates the number of current readers;
- `_cond`, of type `threading.Condition`, which is used as a semaphore.

The `FileByteContent` class uses a semaphore for each open file. In this way, parallel reads and writes to different files are possible, but not on the same file. Methods `_r_acquire()` and `_r_release()` permit respectively to acquire and release the read lock, which can be shared between multiple threads, while methods `_w_acquire()` and `_w_release()` are the equivalent for the write lock, which guarantees exclusive access to the file content.

The acquisition of a lock for reading increases the `_readers` variable, which is decremented at every release. When value 0 is reached, any thread waiting for a write lock is notified. When asking for a write lock on a file, the thread waits as long as there are readers, i.e., if `_readers` is greater than zero. When all the readers have released the resource, the lock is acquired and then released at the end of the write operation.

2.3.6 The *threading* library

The *threading* library [Pyt] is a Python built-in library that supports the management of high-level threads and provides tools for creating and using semaphores. The classes used for read locks and write locks in the `FileByteContent` class are:

- `threading.Lock`, which offers the implementation of the concept of mutex, a mutually exclusive semaphore;
- `threading.Condition`, which represents a condition associated with a particular Lock instance.

In the code of the `FileByteContent` class there is the `_cond` property of type `Condition`, which provides the following methods:

- `acquire`: acquires the underlying lock;
- `release`: releases the underlying lock;
- `wait`: suspends the calling thread and releases the underlying lock, which must have previously been acquired by the thread;
- `notify_all`: wakes up all queued threads, which leave the wait condition only when they gain control over the lock.

The semaphores provided by `Lock` are objects that can be in a state *locked* or *free*. When the lock is in the *free* state, it can be acquired through the `acquire` method, which puts it in the *locked* state. Instead, if `acquire` is invoked while the semaphore is locked, the calling thread pauses until a `release` call frees the mutex.

Objects instantiated by `Condition` are more flexible than simple mutexes, but they are always associated with a semaphore. Through these objects it is possible to suspend a thread with the `wait` method, placing it in a wait queue and causing the underlying lock to be released, and awaken it with `notify` or `notify_all`. Upon exiting the queue, the thread acquires the lock again and then continues in its execution. In this way, it is possible to avoid *busy waits*, that is, situations in which the thread continuously checks the state of the semaphore, occupying computational resources that could be used by other unblocked threads.

Finally, both classes support the *context management protocol*, that is, the use of the Python `with` construct. When the process or thread enters a block delimited by `with`, the `acquire` method of the lock or condition is called, while at the exit the `release` method is invoked.

2.4 Experiments

The virtual file system was tested on a system with Ubuntu 20.04 LTS Focal Fossa, Intel Core i5-7200U quad-core 2.5 GHz CPU, 8 GB RAM, 120GB solid state drive.

The opening of the same file with multiple applications has been tested on the following file formats: text file `.txt`, with *gedit* and *Visual Studio Code*; `.md` text file, with *gedit* and *Typora*; `.jpg` binary files, with *Image Viewer* and *Firefox*; `.pdf` files, with *Document Viewer* and *Xournal*.

For `.txt` files, both *gedit* and *Visual Studio Code* open the file and immediately close it again after reading the entire content. Only when the save operation is invoked, the file is opened again to make the necessary updates. This reduces the possibility of running into concurrency problems that may arise when two applications keep the same file open at the same time. The same situation occurs with `.md` files and `.jpg` files. Applications react differently to changes to the content on disk of an opened file: for example, *gedit* always proposes to the user to reload the entire file, while *Typora* and *Visual Studio Code* transparently update the file and notify the user only if there are pending changes created by the user. The applications used to test *.pdf* files behave differently: both *Document Viewer* and *Xournal* keep the file open and only release it when it is closed. Specifically, *Xournal* reads the content partially as the user scrolls through the document. For this reason, it is necessary to use the counters for open files discussed earlier, to avoid anomalies.

2.4.1 Performance results

The execution times were measured to estimate the performance of some common operations, computed on the basis of 150 tests. To automate the measurements, we used dedicated Python scripts. The files needed for testing are text files created randomly using the command `base64 /dev/urandom | command head -c BYTES> FILENAME.txt`. In Python, `msnow` is a function that returns the Unix timestamp in milliseconds.

Write times are measured by overwriting the entire file: for one file of size D bytes, the bytes written to disk are exactly D. Tables 2.3, 2.4, 2.5 show the mean and standard deviation of the execution times of some common operations, obtained through the scripts, comparing *FreyaFS* in single-thread mode with the direct use of the operating system.

The files are decrypted upon opening and the content is then kept in memory until each file is closed. The reading then takes place on data present in RAM and not from disk, thus offering almost immediate access. In the measurements of the operating system, however, the opening

File size	μ_{FreyaFS} $[ms]$	σ_{FreyaFS} $[ms]$	μ_{OS} $[ms]$	σ_{OS} $[ms]$
4 KB	13.57	0.69	0.01	0.11
1 MB	13.60	0.52	0.01	0.11
10 MB	71.05	2.04	0.01	0.11

Table 2.3: Opening times

File size	μ_{FreyaFS} $[ms]$	σ_{FreyaFS} $[ms]$	μ_{OS} $[ms]$	σ_{OS} $[ms]$
4 KB	0.11	0.35	0.01	0.11
1 MB	1.22	0.54	0.38	0.79
10 MB	8.85	0.84	5.21	2.38

Table 2.4: Reading times

File size	μ_{FreyaFS} $[ms]$	σ_{FreyaFS} $[ms]$	μ_{OS} $[ms]$	σ_{OS} $[ms]$
4 KB	0.14	0.30	0.01	0.08
1 MB	69.08	8.97	0.26	0.44
10 MB	12203	95	4.54	0.62

Table 2.5: Writing times

appears to be good approximation always the same and much less than the opening with *FreyaFS*. This happens because with *FreyaFS* opening also involves reading each of the fragments and decrypting them, operations that slow down execution.

Analyzing the performance, reading turns out to be quite faster than writing. This behavior is due to the basic structure of the All-Or-Nothing-Transform (AONT) protection applied by the virtual file system. Write operations may require to completely rewrite the encryption structure and then force a complete reorganization of the file.

3. AONT techniques for storage protection

The innovation presented in this chapter is an extension of the technique for the application of protection on stored data based on the use of the Mix&Slice technique (adapted in Deliverable D4.2 for its use in Distributed Cloud Storage architectures). This technique was in the previous chapter used to support in a transparent way data protection, integrating its services with the interface of a generic file system. In this chapter, we describe how the internal structure of the protection can be implemented in a more flexible way. The advancement also offers a significant performance benefit in scenarios where hardware acceleration for the application of AES is not available.

3.1 OAEP Mixing

The structure presented in Figure 3.1, which implements an AONT transformation with the use of a classical block encryption algorithm and was presented in Deliverable D4.1, has been evolved, considering the adaptation of the well-known *Optimal Asymmetric Encryption Padding* (OAEP) [BR95, BDPR98] scheme, originally proposed by Bellare and Rogaway. This scheme provides strong inter-dependency in the representation of a resource and was introduced to mitigate the regularity that characterizes the use of asymmetric encryption techniques like RSA. With the application of OAEP before RSA encryption of the plaintext representation of a resource, these regularities are removed.

3.1.1 Feistel network

Feistel networks are a classical solution that was proposed to build block encryption algorithms. The structure of Feistel networks can be traced back to a proposal by Shannon, who suggested the construction of a robust cryptographic algorithm by the repeated application of a transformation that combined *confusion* and *diffusion*. Confusion is obtained by the use of a function, called *round*

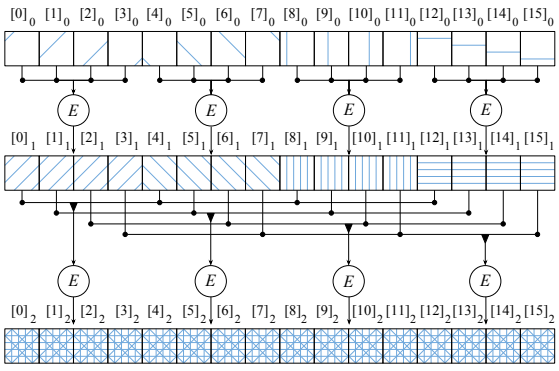

Figure 3.1: Mixing structure presented in Deliverable D4.1

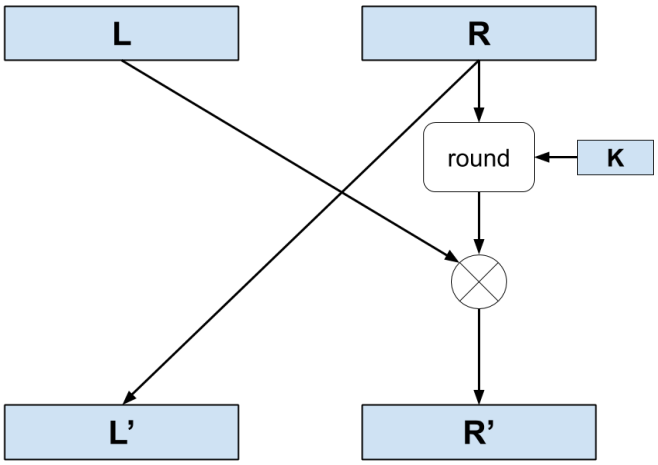

Figure 3.2: Feistel network

function, that breaks the correspondence between the input plaintext and the output ciphertext. Diffusion requires that all the bits in the input block contribute to the result of the transformation, in order to increase the difficulty of any attack that uses the frequency of symbols. The most well-known representative of these algorithms is the Data Encryption Standard (DES). Other well-known block algorithms based on Feistel architecture are Blowfish and TwoFish.

The requirements that have to be satisfied in the construction of a symmetric algorithm operating on blocks, each B-bit long (value B must be even for the application of the Feistel network, as at each iteration the block is divided into two halves of equal size), are that the algorithm must generate a permutation, for all the possible 2^B inputs. The permutation must be driven by the key k and can be inverted by the application of the corresponding decrypting algorithm using the same k. The strength of the protection offered by encryption depends on the fact that the transformation is hard to analyze if the key k is unknown, even with perfect knowledge about the algorithm used (Kerckhoffs' Principle).

The Feistel network achieves these results with the use of an iterative structure, where each step of the iteration is the one described in Figure 3.2. The block is split into two semi-blocks, *Left* (L) and *Right* (R), each half the size of the complete block. The right semi-block R is directly mapped to the left semi-block of the output L', without any modification. The right semi-block R is also provided as input to the *round* function, together with a key k. The output of this function has the same size as the semi-block and it is combined with a XOR with the left semi-block L, producing what will be the right semi-block of the output R'. The output block of each iteration is the input block for the next. In Figure 3.3 we show the repeated application of the structure. We have three iterations in the figure. Real cryptosystems apply a larger number of iterations. For example, DES applies 16 iterations [US 93].

The core of the security resides in the round function, which is required to be a deterministic function that produces an output that is hard to correlate with the input. The round function does not have to be invertible, as it is used in the same way for both encryption and decryption. Looking at the structure of the Feistel transformation, it is relatively easy to verify that the inverse transformation, reported in Figure 3.4, is able to produce the correct inversion. Indeed, given any sequence of B bits as input to the original transformation, for the properties of the binary XOR operator the

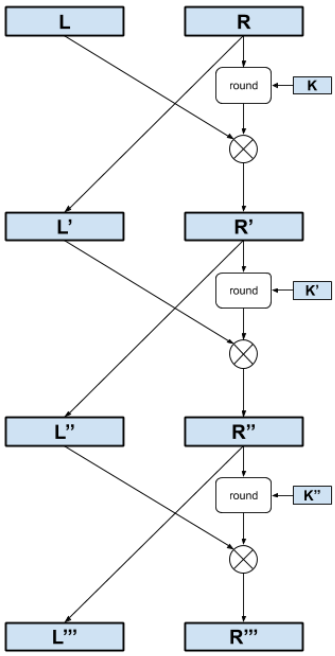

Figure 3.3: Iterative application of a Feistel network

application of the inverse transformation with the same key k will give back the original input. In an iterative application of the structure, possibly with different keys at each iteration, it is sufficient to apply in the inverse order each single transformation to invert the complete application. At each iteration, the protection is actually applied only to half of the input.

The analysis of the features of a Feistel network was produced by Luby and Rackoff [LR88] (this is why the Feistel network is also known as a Luby-Rackoff cipher). Ideally, the round function should behave as a perfect deterministic generator of unpredictable values. If the round function satisfies these requirements, the Feistel construction has been proved to be an ideal cryptographic structure, as long as at least 4 iterations are applied. Most concrete applications of the Feistel architecture in cryptographic algorithms adopt a number of iterations greater than 4 to

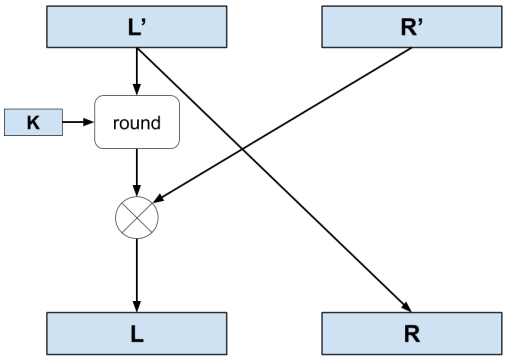

Figure 3.4: Inversion of a Feistel network

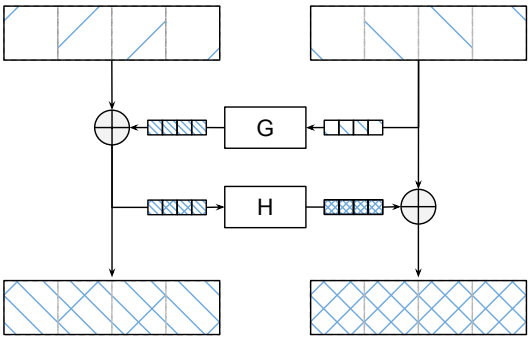

Figure 3.5: Classical OAEP, and the uneven mix of the plaintext

increase the safety margin against the fact that the round function is not ideal.

One of the major advantages of the Feistel network is its simple design and the fact that the security of the encryption derives only from the properties of the round function. As long as the function is well designed and well approximates the behavior of an ideal function, the construction is proved to be secure.

3.1.2 OAEP

Optimal Asymmetric Encryption Padding (OAEP) can be considered as a variant of Feistel network. OAEP uses a pair of cryptographically one-way trapdoor functions, G and H, to process the plaintext prior to asymmetric encryption. Figure 3.5 shows the structure that characterizes OAEP.

Many asymmetric cryptography schemes present regularities in the transformation that they apply to the plaintext. For instance, RSA uses modular exponentiation for the application of both the public and private key. Then, if an adversary has access to the result of the application of the private key d to 2 plaintext messages M_1 and M_2, i.e., $C_1 = (M_1)^d$ and $C_2 = (M_2)^d$, it would be possible to determine the result of the application of the same transformation to the message M_{12} obtained as a product of the two messages $M_{12} = M_1 \cdot M_2$ by simply computing the product of the results of the two applications, because $C_{12} = (M_{12})^d = (M_1 \cdot M_2)^d = (M_1)^d \cdot (M_2)^d = C_1 \cdot C_2$. This makes RSA immediately vulnerable to a chosen-ciphertext attack. With the preliminary application of an OAEP transformation, before a message is transformed using RSA, this regularity is removed. Each message to be encrypted is first processed through an OAEP transformation, which is designed to produce a representation of the original message content in a way that is efficient and easily inverted.

Boyko proved that OAEP can be used to build an All-or-Nothing Transform resistant to chosen-plaintext attacks [Boy99]. The fact that OAEP is an AONT makes it an obvious candidate for its use in the mixing phase of Mix&Slice.

In the concrete implementations of OAEP, functions G and H are always realized with cryptographic hash functions. In the PKCS#1 standard, which prescribes the use of OAEP, functions G and H are identical.

3.2 Use of OAEP for Mixing

Since OAEP implements an AONT, it can be considered as an alternative to the use of the Mixing structure based on the layered application of AES. The benefits from the application of OAEP

can be in terms of efficiency and improved flexibility on the definition of the security parameter, represented in the mixing structure by the size of the miniblock.

With respect to efficiency, the computation of hash functions is traditionally more efficient than the computation of symmetric block ciphers. It is also to consider that symmetric block ciphers like AES are today implemented by dedicated circuits within many CPUs. The throughput exhibited by the hardware-accelerated computation of AES is better than the software computation of modern cryptographic hash functions. If CPU architectures evolve with hardware acceleration of hash functions, this may make the use of OAEP an important option for its performance benefits.

A great benefit of the use of OAEP is the greater flexibility it offers on the security parameter. Using AES and the mixing structure, the largest miniblock size is equal to 64 bits. Most modern cryptographic approaches rely on greater security parameters, as the exploration of 2^{64} configurations is deemed feasible for many adversaries. In most application scenarios, we consider the use of AES-based mixing structure and a miniblock size of 32 or 64 bits as appropriate, but some applications may want more flexibility, particularly if this does not have a significant impact on performance.

A first limitation of the use of OAEP as a replacement for the AONT built on the layered AES-based structure is that by construction the input ($M = |G_{out}| + |H_{out}|$, where G and H are the functions used in the structure in Figure 3.5; in the Mix&Slice structure, this also called the *macroblock*) is larger than the block size of common symmetric block ciphers. In fact, the plaintext must be *xor*-ed with the output of functions G and H.

A trivial technique that can be adopted to extend the size of the output of functions G and H to fit plaintexts of any size is to use G_{out} and H_{out} as the output of a stream cipher with initial seeds; the stream ciphers would produce any required number of bits needed for the XOR operation. This approach would produce an AONT of flexible size. Unfortunately, even if efficient, this scheme that leverages stream ciphers does not satisfy the requirements of the use of an AONT in Mix&Slice. In Mix&Slice, after a miniblock has become unavailable, the reconstruction of a resource must be difficult for an adversary who has previous access to the resource and does not want to store the complete resource content. In the scheme that uses stream ciphers, the adversary can store the two seeds producing G_{out} and H_{out}, which are compact in size, allowing an adversary to trivially invert the scheme and break the All-Or-Nothing Transform.

In order to solve this problem, we need G and H to satisfy the following requirements.

Req.1 G (H, respectively) must not have an internal state that can be efficiently stored to reproduce G_{out} (H_{out}, respectively).

Req.2 Any bit changed in G_{in} (H_{in}, respectively) must have the chance of cryptographically affecting (with a uniform distribution of probability) any bit of G_{out} (H_{out}, respectively), i.e. G and H must be pseudorandom permutations.

Req.3 $|G_{in}| = |G_{out}|$ and $|H_{in}| = |H_{out}|$ must hold, so that an adversary does not have an advantage in storing the input or the output.

Req.1 and *Req.2* imply that G and H have to be what we call *whitebox-AONTs*, representing the property that previous access to the encryption function does not allow to keep a compact representation derived from knowledge of the key or the resource, allowing the revoked user to access the resource after a portion of the resource has been made unavailable. The mixing technique based on the use of AES is a whitebox-AONT and also satisfies *Req.3*, therefore it can be

used as G and H. Moreover, since in the OAEP schema, G and H do not need to be invertible and only have to approximate a pseudorandom function, we can consider the replacement of AES with SHA-512 in the mixing process. The use of SHA-512, which has a block size of 512 bits, permits the reduction of the number of rounds in the mixing process, since it is now possible to mix sixteen 32-bit mini-blocks in one step. Another advantage of the use of SHA-512 as a direct G and H function in the OAEP scheme is that the mini-block size can increase up to 512 bits, for applications that require a stronger protection against brute-force attacks. Considering that in most scenarios where symmetric cryptography is used, a 256-bit size is considered adequate against any adversary relying on extensive computational infrastructures, the ability to go up to a 512-bit miniblock is considered sufficient (for many asymmetric cryptography algorithms, like Diffie-Hellman [DH76] or RSA [RSA78], the size of secret parameters often has to be larger than 512 bits, but these techniques have a different structure and attack options).

It is crucial to note that the regular OAEP construction does not fully satisfy our mixing requirements, in fact OAEP does not imply an *Avalanche Effect* [WT85]: the change of one bit in the left half of the plaintext, impacts the whole right half, but only affects the corresponding bit in the left half, as shown in Figure 3.5. In our scenario, it is necessary to use an adaptation of OAEP that is composed of three rounds, in order to guarantee that the change of any bit in the plaintext has the chance of affecting the whole ciphertext. This result is consistent with the one obtained by Luby and Rackoff in [LR88]. The addition of a fourth round produces a *super* pseudorandom permutation [LR88], which would remain resistant to an adversary who has oracle access to its inverse permutation (not needed to create a whitebox-AONT). This has an impact on the efficiency of the recursive invocation of the scheme, which leads to a higher number of invocations compared to the original mixing structure based on the use of block ciphers.

OAEP does not provide message confidentiality, so after the application of this technique, the output of the OAEP process must be encrypted. We implemented the proposed schema using SHA-512 and *AES* in counter mode (AES-CTR, as specified by NIST [Dwo04]) to encrypt the resource.

The use of the structure then assumes that a resource is stored using this protection structure. Every time there is the need to make the resource unavailable to users who had previously the right to access the resource, a portion at least equal to a miniblock is removed from the resource and, for policy updates, made available to current users encrypted with a fresh key available only to them. Revoked users would then still be able to invert the protection applied by the AES-CTR step, but would be unable to invert the transformation.

3.3 Application of the AONT construction

In general, the use of OAEP gives the flexibility to define a size of miniblock up to 512 bits. We also have to consider how the technique can be adapted to macroblocks of large size. The Mix&Slice technique relies on three major parameters for the configuration of its use: the miniblock size b, the block size B, and the macroblock size M. The miniblock size b represents the minimum number of bits that are changed in each macroblock to forbid access to users who do not have access to the new representation of the miniblock. The macroblock size M defines the size of the minimal data unit used to manage the data. The ratio M/b determines the ideal performance advantage that can be obtained from the use of Mix&Slice in policy update and secure deletion compared to the application of re-encryption and deletion over the whole resource. The block size B depends on the cryptographic function used, e.g., 128 bits for AES. The use of OAEP requires

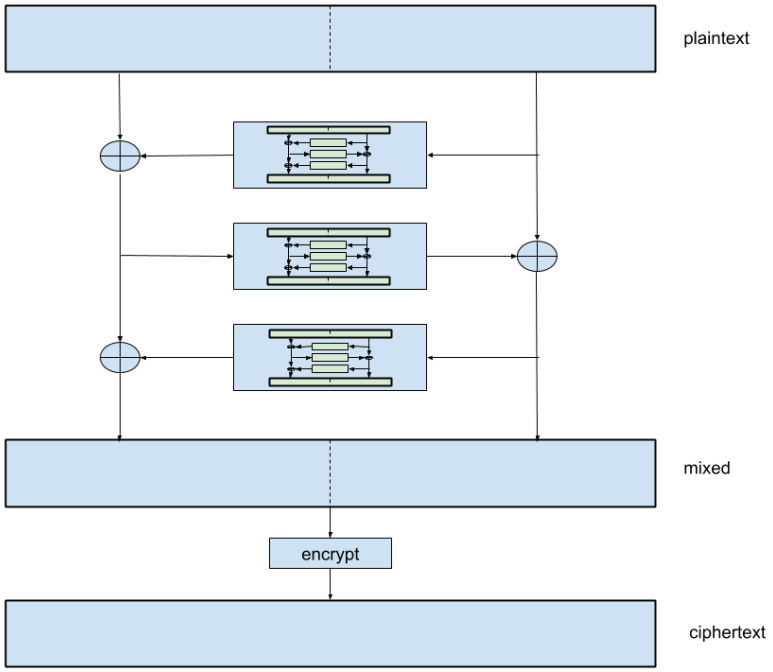

Figure 3.6: Recursive OAEP

the use of secure functions G and H, which we assume to be identical and to be implemented by common cryptographic hash functions, producing outputs of variable size, the largest being 512 bits (i.e., 64 bytes) for SHA-512 and SHA3-512. We will use the name H for the function. When the macroblock size is larger than twice the size of the output of the hash function, a whitebox-AONT construction must be used.

We consider two options for the application of OAEP with macroblocks of size larger than double the size of the hash function. In both cases, we use the constructions that we know are offering the whitebox-AONT property in order to implement the H function. (i) We use at each level the invocation of the OAEP structure, essentially building a fully recursive structure, where OAEP is used at progressively larger sizes. (ii) We use the OAEP structure for the external level, and we implement function H using the layered structure based on the iterated application of a cryptographic function to reach the size of half the macroblock, with the external application of OAEP.

As long as the minimum size of the miniblock is guaranteed by the application of the internal structure, the two alternatives differ on their computational cost. The analysis that we report in Section 3.4 shows that the mixed approach, using for H the layered application of the block function, is more efficient.

3.3.1 Recursive OAEP

The recursive application of OAEP is illustrated in Figure 3.6, where we represent the realization of function H using an OAEP construction operating on a block having size half the size of the original block. This internal OAEP will have to use a function H' operating on a block with size equal to one fourth of the original block B, as it will have to operate over half of its input.

Since the application of OAEP in our construction requires three invocations, the configuration with two levels requires $3^2 = 9$ distinct invocations of function H' in the internal OAEP. It is then

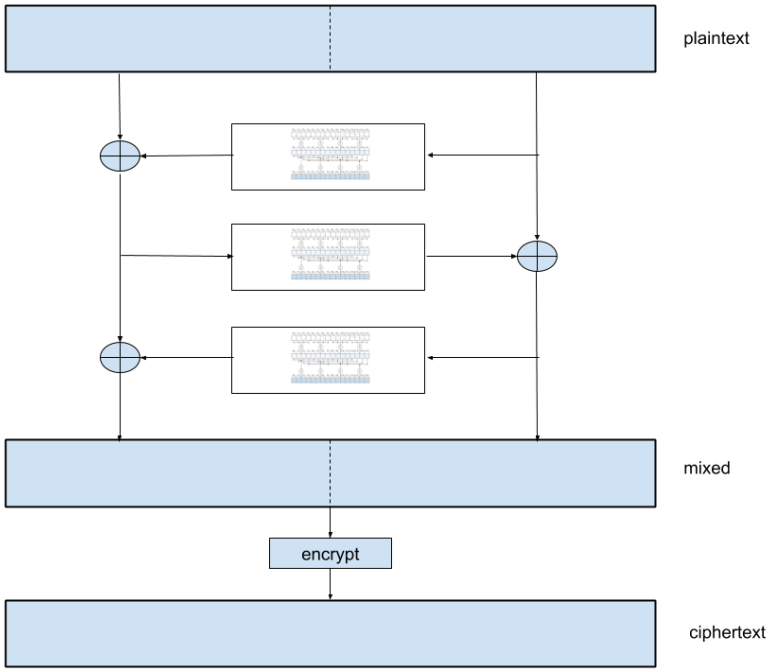

Figure 3.7: OAEP structure with internal layered structure

easy to observe that, assuming that each function H' has size larger than available cryptographic hash functions, in turn this function H' can be implemented using an internal OAEP with a function H'' operating on a block of size $B/8$. This process can be repeated until the block size becomes equal to the size of a cryptographic hash construction. If we assume a size of the cryptographic hash function equal to h and a size of the macroblock equal to M, we will have $\log_2 M/h$ recursion levels, with an overall number of invocations of the hash function equal to $3^{\log_2 M/h}$.

3.3.2 Layered mixing structure

The application of the mixing structure is illustrated in Figure 3.7. The mixing structure is known to offer white-box properties, as long as a robust cryptographic function is used. When the layered structure is used internally to the OAEP, the property of reversibility is not needed, and the functions can be replaced by secure cryptographic functions. Considering that this structure must be applied when the size M of the macroblock is larger than the output of a secure cryptographic function, we can assume to use the largest functions, which are today those producing a 512-bit output. The number of mixing layers depends on the ratio between the size of (half) the macroblock and the size of the miniblock.

The construction we propose is then based on the application of a number of layers l such that $M/2 \leq (B/b)^{l-1}B$. The reduced number of layers and the smaller number of invocations of the cryptographic hash functions makes this a more interesting option compared to the recursive application of OAEP. This option can indeed be quite interesting also in comparison with the use of the original mixing structure. In addition to the greater flexibility in the size of the miniblock, there can be an increase in performance. As we will see in Section 3.4, this is true if we compare the performance of software-based solutions.

Ubuntu 16.04 - Bi-xeon - 256 GB RAM - 1 GiB file - MINI=32bit MACRO=16KB (MINI_PER_MACRO=4096)

	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads
Mix&Slice AES-NI	2.439567	1.416715	0.890562	0.46621	0.358425	0.281926
Mix&Slice 3-passes OAEP (sha512) + aes_ctr	29.392174	17.058399	8.620023	4.814603	3.05236	2.629518
Mix&Slice AES	56.655915	31.374009	17.031515	8.652336	5.790404	5.004548

Ubuntu 16.04 - Bi-xeon - 256 GB RAM - 1 GiB file - MINI=64bit MACRO=16KB (MINI_PER_MACRO=2048)

	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads
Mix&Slice AES-NI	4.027704	2.368083	1.390318	0.751499	0.599889	0.432854
Mix&Slice 3-passes OAEP (sha512) + aes_ctr	29.368057	15.532232	8.279123	4.626541	3.512686	2.565066
Mix&Slice AES	101.946789	62.814606	30.724053	16.609045	10.790716	8.550898

Ubuntu 16.04 - Bi-xeon - 256 GB RAM - 1 GiB file - MINI=128bit MACRO=16KB (MINI_PER_MACRO=1024)

	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads
Mix&Slice AES-NI	-	-	-	-	-	-
Mix&Slice 3-passes OAEP (sha512) + aes_ctr	48.081763	28.664096	15.215565	7.891099	5.361476	4.14707
Mix&Slice AES	-	-	-	-	-	-

Figure 3.8: Seconds required to complete the encryption varying the number of threads

3.4 Experiments

We implemented the OAEP-based mixing structure. The implementation is publicly available within the *Aesmix* library, as an alternative encryption approach (*https://github.com/mosaicrown/aesmix*). We compared its performance profile with the structure based only on the use of AES. The results are presented in Figure 3.8. The table shows that the OAEP-based implementation is faster compared to the AES one that does not leverage the hardware implementation. In fact, even tough the base application of AES-128 and SHA-512 share a similar performance profile, the OAEP approach has a benefit due to the reduction on the number of rounds. However, on architectures that provide a native hardware implementation for *AES* (called *AES-NI*), its use shows the best performance profile for Mix&Slice and should be preferred. As discussed previously, the AES implementation has a maximum mini-block size of 64 bits, whereas the OAEP implementation can be adapted to an arbitrary mini-block size. This is the reason why the third test in Figure 3.8, which assumes a mini-block size greater than 64 bits, only shows the results of the OAEP implementation.

The experiments show that the cost of encryption is compatible with all scenarios for the application of our technique. In particular, the figure illustrates the throughput obtained, varying the number of threads, by the application of our approach in different configurations characterized by macro-blocks of size 16 KiB, mini-blocks of size 32 and 64 bits (which implies 5 and 9 encryption rounds for the mixing structure, resp.), when using AES-NI, when using OAEP, and when using the mixing with AES implemented in software. We notice that even the single-threaded 9-round non-hardware-supported implementation offers a throughput that is greater than 100 Mbps. The experiments also show that, increasing the number of threads, we reach a performance that is more than 10 times the one obtained by the single-threaded implementation. This is consistent with the presence of 12 physical cores in the CPU we used, each with a dedicated AES-NI circuitry.

4. Conclusions

The structure of this deliverable manages two goals, associated with the desire to advance the support of data wrapping in MOSAICrOWN: on one hand we need to have a clear description of how classical techniques are applied, on the other hand it is useful to satisfy the requirements of a research-oriented project by presenting advanced protection techniques and their evolution. The first goal is achieved by Chapter 1, which illustrates the current state of the art for data wrapping in real data markets, clarifying the options that will be considered for the policy-driven application of protection. Chapter 2 and Chapter 3 aim instead at illustrating research work that advances on the state of the art and proposes techniques for the storage of data able to offer strong protection, looking also at the improvement of performance on real-computers.

The focus of this deliverable is the consideration of the use of data wrapping for data protection. In the scenario of digital data markets, protection is also obtained by the use of data sanitization. Data wrapping is realized operating over single data items, relying on the application of transformations that are usually invertible and are based on cryptography, whereas data sanitization requires the consideration of a collection of data items and uses approaches that reduce the information content, by using generalization hierarchies for syntactic approaches, or by introducing a calibrated amount of noise for differential privacy, in both cases producing irreversible transformations. Data wrapping and data sanitization are not alternative, rather they are complementary. Real data markets require the availability of tools supporting both. The overall architecture of MOSAICrOWN and the policy language support the realization of this integration.

Bibliography

[BDPR98] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. In *Advances in Cryptology-CRYPTO'98*, pages 26–45. Springer, 1998.

[Boy99] V. Boyko. On the security properties of oaep as an all-or-nothing transform. In *Proceedings of the 19th International Cryptology Conference (CRYPTO)*, 1999.

[BR95] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In *Advances in Cryptology-EUROCRYPT'94*, pages 92–111. Springer, 1995.

[DH76] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theory*, 22(6):644–654, 1976.

[Dwo04] M. J. Dworkin. Sp 800-38c. Recommendation for block cipher modes of operation: The CCM mode for authentication and confidentiality. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2004.

[FUSa] FUSE Project. Fuse: The linux kernel documentation. `https://www.kernel.org/doc/html/latest/filesystems/fuse.html`.

[FUSb] FUSE Project. fusepy: Python module that provides a simple interface to fuse. `https://github.com/fusepy/fusepy`.

[FUSc] FUSE Project. libfuse: reference implementation for communicating with the fuse kernel module. `https://github.com/libfuse/libfuse`.

[HKN12] J. Heurix, M. Karlinger, and T Neubauer. Pseudonymization with metadata encryption for privacy-preserving searchable documents. *2012 45th Hawaii International Conference on System Sciences*, 2012.

[Jan13] P. Janulek. Tokenization in payment environments. *United States Patent Publication*, 2013.

[LR88] M. Luby and C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM J. Comp.*, 17(2):373–386, Apr. 1988.

[MOS20] MOSAICrOWN project. Aesmix: implementation of the mix&slice encryption mode. `https://github.com/mosaicrown/aesmix`, 2020.

[MR19] U. Mattsson and Y. Rozenberg. Dash: A payments-focused cryptocurrency. *Dashpay*, 2019.

[NH11] T. Neubauer and J. Heurix. A methodology for the pseudonymization of medical data. *International Journal of Medical Informatics*, 80(3):190–204, 2011.

[NS19] K. Nagaraj and A. Sridhar. Encrypting and preserving sensitive attributes in customer churn data using novel dragonfly based pseudonymizer approach. *MDPI*, 10(9), 2019.

[Pyt] Python Software Foundation. threading: Thread-based parallelism in python. `https://docs.python.org/3/library/threading.html`.

[RSA78] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[Tow10] Townsend Security. Tokenization: A cost-effective and easy path to compliance and data protection. *Whitepaper by Townsend*, 2010.

[US 93] US Department of Commerce. Data Encryption Standard (DES). Technical Report FIPS PUB 46-2, US Department of Commerce, December 1993.

[WT85] A. F. Webster and S. E. Tavares. On the design of S-boxes. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 523–534. Springer, 1985.